МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Алгоритмы и структуры данных»
Тема: Линейные структуры данных: стек, очередь, дек
Вариант 11-D-B

Студент гр. 8304 Сани З. Б

Преподаватель Фирсов М. А.

Санкт-Петербург
2019

Цель работы.

Познакомиться с часто используемыми на практике линейными структурами данных, обеспечивающими доступ к элементам последовательности только через её начало и конец, и способами реализации этих структур, освоить на практике использование стека, очереди и дека для решения задач.

Постановка задачи.

Рассматривается выражение следующего вида:

< выражение >::= < терм > | < терм > + < выражение > |
< терм > − < выражение >
< терм > ::= < множитель > | < множитель > * < терм >
< множитель > ::= < число > | < переменная > | (< выражение >) |
< множитель > ^ < число >
< число > ::= < цифра >
< переменная > ::= < буква >

Такая форма записи выражения называется ***инфиксной***.

Постфиксной (префиксной) формой записи выражения *aDb* называется запись, в которой знак операции размещен за (перед) операндами: *abD* (*Dab*).

Примеры

Инфиксная	***Постфиксная***	***Префиксная***
a−b	*ab−*	*−ab*
*a*b+c*	*ab*c+*	*+*abc*
a(b+c)*	*abc+**	**a+bc*
*a+b^c^d*e*	*abc^d^e*+*	*+a*^b^cde*.

Отметим, что постфиксная и префиксная формы записи выражений не содержат скобок.

Вариант 11-D-B: вывести в обычной (инфиксной) форме выражение, записанное в постфиксной форме в заданном текстовом файле postfix (рекурсивные процедуры не использовать и лишние скобки не выводить);

Описание алгоритма.

Для начала программа должна считать данные и передать строку в функцию преобразования выражения из постфиксной формы в инфиксную. Далее для перевода выражения из постфиксной в инфиксную запись необходимо следовать алгоритму:

1) Если отсканированный символ является операндом (цифрой или алфавитом), то мы помещаем его в стек.

2) Если читаем знак операции, то:
   1. Берем текущий знак операции и следующий
   2. Если в первом элементе приоритет операции меньше (и не равен 0), чем у рассматриваем операции, то берем первый элемент в скобки
   3. Аналогично для 2-го элемента
   4. Записываем в стек строку вида: 2-й элемент + знак операции + 1-й элемент

3) Если строка полностью пройдена, то результатом является значение вершины стека

Описание функций и СД.

Класс Stack реализует структуру стека, а также методы для работы с ним. Стандартные методы для работы со стеком:

```
void push(const Data elem);
void pop();
Data top() const;
size_t size() const;
bool isEmpty() const;
```

main.cpp реализует алгоритм перевода выражения из постфиксной формы в инфиксную с помощью стека.

функция для перевода выражения из постфиксной формы в инфиксную:

```
std::string getInfix(std::string postfixExpression);
```

Принимает на вход строку-выражение, возвращает выражение в инфиксной форме, если исходное выражение корректно и пустую строку в случае ошибки. Строка анализируется посимвольно. Если текущий символ "число или буква", тогда элемент помещается в стек с приоритетом 0, если текущий символ "знак операции", из стека достаются два элемента, затем вызывается функция Data **makeInfix**(char sign, Data firstArg, Data second Arg), которая применяет оператор и возвращает форму инфикса, затем мы помещаем его в стек.В ходе преобразования, если стек пустой, выводится ошибка и возвращается пустая строка. После преобразования в стеке должен находиться один элемент - инфиксное выражение.

ПРИЛОЖЕНИЕ

1. Тестирование

Работа программы для выражения A B + C D + *

```
> Choose your input
> 0 - from console
> 1 - from file default file -(default test file is located along the path : test3.txt)
> Any other number to Exit!
0
> Enter expression: A B + C D + *
Result of expression placed in stack :A+B
Result of expression placed in stack :C+D
Result of expression placed in stack :(A+B)*(C+D)
---->   FINAL EXPRESSION : (A+B)*(C+D)

Program ended with exit code: 0
```

Таблица результатов ввода/вывода тестирования программы

Постфиксное выражение	**Инфиксное выражение**
a	a
a b -	a-b
a b * c +	a*b+c
a b c + *	a*(b+c)
a b c ^ d ^ e * +	a+b^c^d*e
b c d e + + *	b*(c+d+e)
a b c d e + + * +	a+b*(c+d+e)
a b c d ^ - *	a*(b-c^d)
r k + x * g ^	((r+k)*x)^g
A B + C D + *	(A+B)*(C+D)
a b + c * d + e f * -	(a+b)*c+d-e*f
2 3 4 5 + * +	2+3*(4+5)
a 2 3 4 ^ - *	a*(2-3^4)
23 67 90 - + 9 ^	(23+67-90)^9

```
> Choose your input
> 0 - from console
> 1 - from file default file -(default test file is located along the path : test3.txt)
> Any other number to Exit!
1
> FilePath:  /Users/sanizayyad/Documents/Sani_Zayyad/lab3/Test/test3.txt
test #1 "a"
---->  FINAL EXPRESSION : a

test #2 "a b -"
Result of expression placed in stack :a-b
---->  FINAL EXPRESSION : a-b

test #3 "a b * c +"
Result of expression placed in stack :a*b
Result of expression placed in stack :a*b+c
---->  FINAL EXPRESSION : a*b+c

test #4 "a b c + *"
Result of expression placed in stack :b+c
Result of expression placed in stack :a*(b+c)
---->  FINAL EXPRESSION : a*(b+c)

test #5 "a b c ^ d ^ e * +"
Result of expression placed in stack :b^c
Result of expression placed in stack :b^c^d
Result of expression placed in stack :b^c^d*e
Result of expression placed in stack :a+b^c^d*e
---->  FINAL EXPRESSION : a+b^c^d*e

test #6 "b c d e + + *"
Result of expression placed in stack :d+e
Result of expression placed in stack :c+d+e
Result of expression placed in stack :b*(c+d+e)
---->  FINAL EXPRESSION : b*(c+d+e)

test #7 "a b c d e + + * +"
Result of expression placed in stack :d+e
Result of expression placed in stack :c+d+e
Result of expression placed in stack :b*(c+d+e)
Result of expression placed in stack :a+b*(c+d+e)
---->  FINAL EXPRESSION : a+b*(c+d+e)

test #8 "a b c d ^ - *"
Result of expression placed in stack :c^d
Result of expression placed in stack :b-c^d
Result of expression placed in stack :a*(b-c^d)
---->  FINAL EXPRESSION : a*(b-c^d)
```

```
test #9 "r k + x * g ^"
Result of expression placed in stack :r+k
Result of expression placed in stack :(r+k)*x
Result of expression placed in stack :((r+k)*x)^g
---->  FINAL EXPRESSION : ((r+k)*x)^g

test #10 "A B + C D + *"
Result of expression placed in stack :A+B
Result of expression placed in stack :C+D
Result of expression placed in stack :(A+B)*(C+D)
---->  FINAL EXPRESSION : (A+B)*(C+D)

test #11 "a b + c * d + e f * -"
Result of expression placed in stack :a+b
Result of expression placed in stack :(a+b)*c
Result of expression placed in stack :(a+b)*c+d
Result of expression placed in stack :e*f
Result of expression placed in stack :(a+b)*c+d-e*f
---->  FINAL EXPRESSION : (a+b)*c+d-e*f

test #12 "2 3 4 5 + * +"
Result of expression placed in stack :4+5
Result of expression placed in stack :3*(4+5)
Result of expression placed in stack :2+3*(4+5)
---->  FINAL EXPRESSION : 2+3*(4+5)

test #13 "a 2 3 4 ^ - *"
Result of expression placed in stack :3^4
Result of expression placed in stack :2-3^4
Result of expression placed in stack :a*(2-3^4)
---->  FINAL EXPRESSION : a*(2-3^4)

test #14 "23 67 90 - + 9 ^"
Result of expression placed in stack :67-90
Result of expression placed in stack :23+67-90
Result of expression placed in stack :(23+67-90)^9
---->  FINAL EXPRESSION : (23+67-90)^9
```

Выводы.

В ходе работы были приобретены навыки работы со стеком, изучены методы работы с ним (объявлять, заносить в него переменных и забирать их). Был изучен и реализован алгоритм перевода записи из постфиксной в инфиксную.

2. Исходный код программы.

stack.hpp

```
#ifndef stack_hpp
#define stack_hpp

#include <string>
#include <iostream>

// A stack element consisting of an expression and the priority of that expression.
// check makeInfix function in main.cpp to see the definition of priorities for every operator.
typedef std::pair<std::string, int> Data;

class Stack{
public:
    Stack(size_t max_size);

    Stack(const Stack &stack) = delete;
    Stack& operator=(const Stack&) = delete;

    ~Stack();

    //methods for working with stack
    void push(const Data elem);
    Data pop();
    Data top() const;
    size_t size() const;
    bool isEmpty() const;

private:
    Data* stackData;
    std::size_t stackSize;

};


#endif /* stack_hpp */
```

stack.cpp

```
#include "stack.hpp"

Stack::Stack(size_t maxSize)
{
   stackData = new Data[maxSize];
   stackSize = 0;
}

Stack::~Stack()
{
    delete [] stackData;
}


//getting size of stack
size_t Stack::size() const
{
   return stackSize;
}

//checking if the stack is empty
bool Stack::isEmpty() const
{
   return size() == 0;
}

//getting the top element of the stack
Data Stack::top() const
{
   if (!isEmpty()) {
      return stackData[stackSize -1];
   }
   else {
      std::cout << "Error. Stack is empty";
      return Data("", 0);
   }
}

// removing or popping the top element in stack
Data Stack::pop()
{
   if (!isEmpty()){
       Data elem = stackData[--stackSize];
      return elem;
```

```
    }
    else {
        std::cout << "Error. Stack is empty";
        return Data("",0);
    }

}

//adding element in stack
void Stack::push(const Data elem)
{
    stackData[stackSize] = elem;
    stackSize++;
}
```

main.cpp

```
#include "stack.hpp"
#include <iostream>
#include <fstream>
#include <string>

void ReadFromFile(std::string filename);
std::string getInfix(const std::string &inputt);
Data makeInfix(char sign, Data firstArg, Data secondArg);
bool isAlpha(const char ch);
bool isDigit(const char ch);
bool isOperator(const char ch);


void ReadFromFile(std::string filename)
{
    std::ifstream file(filename);

    if (file.is_open())
    {
        int count = 0;
        while (!file.eof())
        {
            count++;
            std::string input;
            getline(file, input);
```

```
            std::cout << "test #" << count << " \"" + input + "\""<< "\n";
            std::string output = getInfix(input);
            std::cout<< "---->  FINAL EXPRESSION : "<< output <<"\n\n";
        }
    }
    else
    {
       std::cout << "File not opened"<<"\n";
    }
}


std::string getInfix(const std::string &postfixExpression){
    Stack stack;
    for (auto i = postfixExpression.cbegin(); i < postfixExpression.end(); ++i) {
      char elem = *i;
       if(elem == ' '){
           continue;
       }
       // If the scanned character is an operand (number||alpahabet),
       // push it to the stack.
      else if (isAlpha(elem) || isDigit(elem)) {
           std::string tmpStr;
           while (elem != ' ' && i != postfixExpression.end()) {
             if (!isDigit(elem) && !isAlpha(elem)) {
                 std::cout<< "Error: wrong data!";
                return "";
             }
             tmpStr += elem;
             ++i;
             elem = *i;
           }
           stack.push(Data(tmpStr, 0));
       }
      // If the scanned character is an operator, pop two
       // elements from stack apply the operator
       else if (isOperator(elem)) {
           Data firstArg;
           Data secondArg;

           if (!stack.isEmpty()) {
              secondArg = stack.top();
              stack.pop();
           }
```

```
            else {
               std::cout << "Error: stack is empty!";
               return "";
            }

            if (!stack.isEmpty()) {
               firstArg = stack.top();
               stack.pop();
            }
            else {
               std::cout << "Error: stack is empty!";
               return "";
            }

            //applying the operator
            Data infix = makeInfix(elem, firstArg, secondArg);
            //then pushing it to stack
            stack.push(infix);
            std::cout << "Result of expression placed in stack :" << infix.first.c_str()<< "\n";
         }
         else {
            std::cout<< "Incorrect symbol in string!";
            return "";
         }
      }

      // There must be a single element
      // in stack now which is the required infix.
      if (stack.size() == 1) {
         std:: string tmpStr = stack.top().first;
         stack.pop();
         return tmpStr;
      }
      else {
         std::cout << "Error: string is incorrect!";
         return "";
      }

}
Data makeInfix(char sign, Data firstArg, Data secondArg){
   std::string tmpStr;

   int expressionPriority = 0;
    //defining the priority for each sign(operator) according to PEDMAS
```

```
    //exponent > multiplication > adittion and subtraction
    if (sign == '+' || sign == '-') {
        expressionPriority = 1;
    } else if (sign == '*') {
        expressionPriority = 2;
    } else if (sign == '^') {
        expressionPriority = 3;
    }

    if (firstArg.second != 0 && firstArg.second < expressionPriority) {
        tmpStr += '(';
        tmpStr += firstArg.first;
        tmpStr += ')';
    } else {
        tmpStr += firstArg.first;
    }

    tmpStr += sign;

    if (secondArg.second != 0 && secondArg.second < expressionPriority) {
        tmpStr += '(';
        tmpStr += secondArg.first;
        tmpStr += ')';
    } else {
        tmpStr += secondArg.first;
    }

    Data tmp(tmpStr,expressionPriority);

    return tmp;
}

//check if element is an operand (in this case Alphabets)
bool isAlpha(const char ch)
{
    return ((ch >= 'a' && ch <= 'z') ||
            (ch >= 'A' && ch <= 'Z'));
}
//check if element is an operand (in this case numbers)
bool isDigit(const char ch)
{
    return (ch >= '0' && ch <= '9');
}
```

```
//check if element is an operator
bool isOperator(const char ch)
{
   return (ch == '+' || ch == '-' ||
           ch == '*' || ch == '^');
}
```

```
int main()
{
   std::cout << "> Choose your input" << "\n";
   std::cout << "> 0 - from console" << "\n";
   std::cout << "> 1 - from file default file -(default test file is located along the path : test3.txt)" << "\n";
   std::cout << "> Any other number to Exit!" <<"\n";

   int command = 0;
   std::cin >> command;
   std::cin.ignore();

   switch (command)
   {
      case 0:
      {
         std::cout << "> Enter expression: ";
         std::string input,output;
         std::getline(std::cin, input);
         output = getInfix(input);
         std::cout<< "---->   FINAL EXPRESSION : "<< output <<"\n\n";
         break;
      }
   case 1:
      {
         std::cout << "> FilePath: ";
         std::string filePath;
         std::cin >> filePath;
         ReadFromFile(filePath);
         break;
      }
   default:
         std::cout << "GOODBYE!";
```

```
    }
    return 0;
}
```